Naval Research Laboratory
Stennis Space Center, MS 39529-5004

NRL/CR/7323--96-0006
Contract: N0014-92-D-6008

A Scalable Implementation of the NRL Layered Ocean Model

ALAN J. WALLCRAFT
DANIEL R. MOORE

Planning Systems Incorporated
Slidell, LA

August 14, 1998

REPORT DOCUMENTATION PAGE

Form Approved
OBM No. 0704-0188

1. Agency Use Only *(Leave blank).*	2. Report Date.	3. Report Type and Dates Covered.
	August 14, 1998	Contractor Report

4. Title and Subtitle.

A Scalable Implementation of the NRL Layered Ocean Model

5. Funding Numbers.

Contract	N0014-92-D-6008
Program Element No.	0602435N
Project No.	
Task No.	03523
Accession No.	DN 164222
Work Unit No.	73-7228-00

6. Author(s).

Alan J. Wallcraft* and Daniel R. Moore*

7. Performing Organization Name(s) and Address(es).

Naval Research Laboratory
Ocean Dynamics and Prediction Branch
Oceanography Division
Stennis Space Center, MS 39529-5004

8. Performing Organization Report Number.

9. Sponsoring/Monitoring Agency Name(s) and Address(es).

Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217-5660

10. Sponsoring/Monitoring Agency Report Number.

NRL/CR/7323--96-0006

11. Supplementary Notes.

*Planning Systems Incorporated, 115 Christian Lane, Slidell, LA 70458

12a. Distribution/Availability Statement.

Approved for public release; distribution unlimited.

12b. Distribution Code.

13. Abstract *(Maximum 200 words).*

This report describes the portable scalable implementation of the NRL Layered Ocean Model (NLOM). Scalability is based primarily on the tiled data parallel, parallel programming paradigm. This is sufficiently general that the actual technique used on a given machine to obtain scalability can be selected at compile time from: (i) data parallel, (ii) SPMD message passing, (iii) autotasking, or (iv) SPMD message passing between multi-processor autotasked systems. The code is thus portable onto all machine types likely to be used by ocean modelers.

14. Subject Terms.

ocean model, scalable, data parallel, message passing, autotasking

15. Number of Pages. 34

16. Price Code.

17. Security Classification of Report.	18. Security Classification of This Page.	19. Security Classification of Abstract.	20. Limitation of Abstract.
Unclassified	Unclassified	Unclassified	SAR

CONTENTS

EXECUTIVE SUMMARY E-1

1. INTRODUCTION 1

2. DISTRIBUTED MEMORY PROGRAMMING STYLES 2

2.1 Data Parallel 2
2.2 Tiled Message Passing 3
2.3 Tiled Data Parallel 5

3. NLOM IMPLEMENTATION 6

3.1 Example of Autotasking Code 6
3.2 Example of Tiled Data Parallel Code 7
3.3 Communication Subroutines 11
3.4 Parallel I/O 18
3.5 Serial Optimizations 19
3.6 Cache Management 21
3.7 Parallelizing the Capacitance Matrix Technique 23

4. NLOM PERFORMANCE 25

5. SUMMARY 30

6. ACKNOWLEDGEMENTS 30

7. REFERENCES 30

EXECUTIVE SUMMARY

This report describes the portable scalable implementation of the NRL Layered Ocean Model (NLOM). Scalability is based primarily on the tiled data parallel parallel programming paradigm. This is sufficiently general that the actual technique used on a given machine to obtain scalability can be selected at compile time from: (i) data parallel, (ii) SPMD message passing, (iii) autotasking, or (iv) SPMD message passing between multi-processor autotasked systems. The code is thus portable onto all machine types likely to be used by ocean modelers.

Techniques generally applicable to ocean models on distributed memory machines are first introduced using a simple example. This is extended to a practical case by examining the actual source code for one subroutine from the original (autotasked) and scalable NLOM implementations. These are very similar, because distributed memory aspects are encapsulated in a suite of communication subroutines. The three versions (data parallel, autotasked, message passing) of a representative communication subroutine are presented and discussed. The original NLOM implementation was primarily optimized for vector parallel systems. Several new optimizations have been added to the scalable implementation to allow for the relatively small cache and low memory bandwidth of RISC-based systems.

The high performance of NLOM depends critically on the use of a semi-implicit time step, and solving the consequent external gravity mode Helmholtz's equation using the Capacitance Matrix Technique (CMT). The scalable CMT implementation is described, including the "pipelining" method used to obtain scalability without sacrificing numerical repeatability no matter how many processors are involved.

There are currently no standard methods for implementing general parallel I/O. The scalable NLOM code uses separate files for "scalar" and "array" data. This allows a degree of parallel I/O to array files on some machines, while preserving full portability between machines.

The scalable NLOM implementation has achieved its design goals of running exactly the same model source code and model data files on a wide range of computer systems from many vendors. Times from actual practical model runs presented in this report demonstrate good scalability across the range of processor complexes available within DoD today. As larger machines become available, problem sizes (i.e. model resolution) will increase and scalability should be possible up to at least 2048 processors.

A Scalable Implementation of the NRL Layered Ocean Model

1. INTRODUCTION

The NRL Layered Ocean Model, NLOM, has been under continuous development for 20 years [8], [15], [12]. The Users Guide [15] describes the model at the time the guide was released, and is still the primary reference for the model formulation. A new implementation of horizontal diffusion and a discussion of the model formulation in spherical coordinates can be found in Moore and Wallcraft [12]. Here we describe the techniques used to make the model's implementation (source code) portable to all major computer architectures and to several scalable programming styles. Many of the techniques described will be applicable to other ocean models.

The original implementation was targeted to SMP (shared memory parallel) systems, and in particular to PVP (parallel vector processors) where each CPU is a vector processor. This was the dominant supercomputer architecture throughout the 1980's, and is still used in machines such as Cray's C90 today. The model was written in standard Fortran 77 using a coding style that allowed compilers to automatically run loop nests in parallel across multiple processors, while still automatically vectorizing inner loops. The compiler technology involved is known as autotasking in Cray compilers. It is available in the Fortran compilers from all major vendors of SMP systems, and this has allowed the original ocean model implementation to run efficiently on a wide range of shared memory machines, from workstations to supercomputers.

SMP systems are limited by the need to have uniform access from each processor to each element in memory, which is only cost effectively implementable for a maximum of 16, or perhaps 32, processors. Distributed memory architectures make memory local to the processors. This allows scalability to hundreds of processors, but at the expense of all or part of hardware support for the uniform shared memory programming style. On non-uniform memory access (NUMA) systems, all memory is accessible (without remote processor involvement) from all processors, but access speeds are faster (perhaps much faster) for local memory. Many distributed systems do not implement a shared memory at all, requiring messages to be transfered over a network to access another processor's memory typically with the cooperation of the remote processor. It has so far proved impossible, in general, to take existing "dusty deck" shared memory source code and have a compiler automatically produce an efficient distributed memory version. This is the case even if the original source was optimized for SMP systems and the target is a cluster of SMP machines with a NUMA global shared memory, and these are the most similar shared and distributed memory designs. The problem is that compilers are best at local optimizations (e.g. DO-loop or subroutine level), but efficient use of a distributed memory requires global decisions about how memory is used.

Since distributed memory systems are almost certain to be the fastest available for ocean

Manuscript approved September 4, 1997.

modeling over the next 10 years, an implementation of NLOM that can run efficiently on these machines has been produced. Just as the original vector/parallel implementation ran with reasonable efficiency on any single processor or SMP system, an important design goal for the scalable implementation is that it be reasonably efficient on any computer system likely to be used for ocean modeling: workstations, workstation clusters, SMP systems, SMP clusters, vector, vector-parallel, and massively parallel computer systems. This allows a single source code to be used across all machine types, greatly simplifying maintenance and the development of new features.

2. DISTRIBUTED MEMORY PROGRAMMING STYLES

2.1 Data Parallel

One approach for distributed memory machines that retains the idea of a global shared memory is data parallel programming. High Performance Fortran (HPF) is an extension to Fortran 90 implementing the data parallel approach [9]. It consists of a standard set of functions, compiler directives, and a very few language extensions. The language extensions have been included in Fortran 95, so a HPF program is also a Fortran 95 program (i.e. is portable to any machine with a Fortran 95 compiler). HPF is based on CM Fortran, which is a data parallel language for the CM5. The first distributed memory implementation of NLOM was written in CM Fortran, and could also run on any machine with a Fortran 90 compiler. The code was completely different from the original SMP Fortran 77 version, because CM Fortran requires all parallel operations be expressed using Fortran 90 array expressions. CM Fortran also requires the addition of directives that tell the compiler how to layout arrays across the distributed memory. These directives do not change the meaning of the program, but they have a very large effect on performance. Optimization for the CM5 (or any data parallel machine/language) is largely involved with getting memory layout right. In principle, the required array syntax could be automatically generated from the Fortran 77 model code. However, the current generation of tools cannot do this, despite the basic self similarity of finite difference codes. Consider the following simple code fragment:

```
REAL    A(IH+1,JH),DA(IH+1,JH),DX
INTEGER I,J
DO J= 1,JH
  DO I= 1,IH
    DA(I,J) = DX*(A(I+1,J) - A(I,J))
ENDDO; ENDDO
```

The arrays `A` and `DA` have been extended by a one column "halo" to allow a clean implementation of a periodic boundary. On entry `A(IH+1,:)` must be identical to `A(1,:)`, and at some later phase in the program this condition would be explicitly applied to `DA` also. The equivalent CM Fortran is:

```
      REAL A(IH,JH),DA(IH,JH),DX
CMF$  LAYOUT A(:NEWS,:NEWS),DA(:NEWS,:NEWS)
      DA = DX*(CSHIFT(A,SHIFT=+1,DIM=1) - A)
```

Automatic converters might be able to handle this simple example, but in an actual code the use of a halo to handle periodic boundaries would not be correctly converted to a `CSHIFT` on an array without a halo. The arrays `A` and `DA` are no longer extended by one column, because the `CSHIFT` (periodic, or circular, shift) array operation is a one to one mapping to periodic boundary conditions.

The `LAYOUT` directive distributes each 2-D array across a 2-D grid of processors, i.e. if the grid is `MP` by `NP`, the `M,N` processor contains the sub-array `A( MO+1:MO+IHP, NO+1:NO+JHP )` where `IHP=IH/MP`, `JHP=JH/NP`, `MO=(M-1)*IHP` and `NO=(N-1)*JHP`. Under the data parallel approach, the owner of the assigned value calculates that value and off-chip values are copied as required. Thus the `CSHIFT` operation involves communication, and an advantage of using `CSHIFT` or `EOSHIFT` in a data parallel code is that it makes this communication explicit. The data parallel programming model works as if all non-array operations are performed on one processor and the result broadcast to all others. A data parallel compiler can have each processor simultaneously calculate such values instead, providing the effect is the same as broadcasting the result from a single processor.

The data parallel approach is similar to autotasking, in that parallelization is a local process. This implies that autotasking compilers could certainly be extended to parallelize arbitrary array expressions. Therefore, HPF (or CM Fortran) codes are in principle portable to almost any machine design. The CM Fortran implementation of NLOM is in fact a legal Fortran 90 code, so it can be run on a Cray C90 (say) or a workstation with a Fortran 90 compiler. The problem with using HPF for a portable code is efficiency:

- HPF requires communication for almost every array expression.
- Different LAYOUT configurations may be optimal on different machines, i.e. difficult to write portable optimized HPF.
- Data parallel codes tend to use many array temporaries.
- Data parallel codes are typically written to be scalable to a large number of processors. This almost always involves additional floating point operations, and these reduce performance on machines with a small number of processors.
- A single nested DO-loop is typically equivalent to several array operations, reducing optimization opertunities.
- Current Fortran 90 compilers produce better code for DO loops than for array expressions.

Some of these limitations will be significantly reduced as compilers improve. On SMP machines, a smart compiler might eventually be able to make a collection of data parallel array expressions as efficient as a similar set of nested DO loops. However, the data parallel version will probably still require more memory. On distributed memory machines, a simplistic HPF compiler only works well if remote memory access is very fast (as it can be on a data parallel machine such as the CM5, or on most NUMA systems). More sophisticated compilers can try to cache selected remote memory values to reduce network traffic.

After producing a working data parallel version of NLOM [14], it became clear that it was too slow and memory hungry to replace the existing version on SMP systems. The data parallel version worked well on the CM5, but maintaining two separate versions of an ocean model that is itself under continuous development was a serious maintenance headache. Moreover, HPF compilers for new distributed memory systems (such as the T3D) were incomplete and not sufficiently robust to allow easy and early migration to each new system.

2.2 Tiled Message Passing

Message passing libraries can be considered the assembly language of distributed memory machines. For ocean models, message passing is typically used via the domain decomposition SPMD

(Single Program Multiple Data) programming style. The domain is divided into approximately equally sized pieces, one per processor, and each processor runs the same program over its piece of the domain, calling a message passing library to obtain values from other sub-domains. All scalar calculations, i.e. those not acting directly on model domain fields, are calculated separately on each processor. This implies that in practice SPMD must use a homogeneous set of processors, otherwise the scalar calculations could diverge. Efficiency and program transparency is enhanced by including a halo (or fake zone, or ghost points) of nearby points, which allows existing finite difference code to be used almost without change provided the halo is updated, via message passing, at appropriate intervals. Consider our simple code fragment:

```
REAL    A(IH+1,JH),DA(IH+1,JH),DX
INTEGER I,J
DO J= 1,JH
  DO I= 1,IH
    DA(I,J) = DX*(A(I+1,J) - A(I,J))
ENDDO; ENDDO
```

To convert to domain decomposition, add a halo on all sides and have the array otherwise extend over the subdomain only:

```
REAL    A(0:IHP+1,0:JHP+1),DA(0:IHP+1,0:JHP+1),DX
INTEGER I,J
DO J= 1,JHP
  DO I= 1,IHP
    DA(I,J) = DX*(A(I+1,J) - A(I,J))
ENDDO; ENDDO
```

A 2-D, `MP` by `NP`, grid of processors are all running this identical program, with `IHP=IH/MP` and `JHP=JH/NP`. Provided the halo is up to date, the code fragment calculates the required values over the subdomain owned by this processor. Note that no knowledge is required about which subdomain is being calculated. This is all handled by the message passing and I/O routines which would need to be added to the original code.

This is actually a special case of domain decomposition, *tiling*, that is applicable to all finite difference ocean models. The sub-regions are rectangular and halos have a simple mapping onto nearby processors. If the tiles are all the same size (as they are here) this gives a mapping of arrays to processors that is very similar to the data parallel approach. Data parallel codes don't have to explicitly include a halo, but halos (invisible to the programmer) are one technique a data parallel compiler can use to cache frequently accessed off-processor memory locations.

Converting an existing autotasked code to use the tiled message passing approach has a minimal effect on SMP performance, since when compiled for a single tile covering the entire region the original code is effectively used except for a few calls to update the tile halos. Note that in the single tile mode, halo updates do not involve message passing and can include periodic boundary conditions. However, if an existing code cannot autotask efficiently, then the only SMP parallelization option with this approach is to message pass between multiple tiles on the same SMP machine, and current message passing libraries are not efficient when running in a multi-user SMP environment.

Overall, tiled message passing is a viable approach to portability. Certainly more viable today than the data parallel approach. It does have some disadvantages:

- Message passing is inefficient on data parallel machines, such as the CM5.
- All non-local operations require a subroutine call, e.g. `AIJ = A(I,J)` becomes `CALL XCEGET(AIJ, A,I,J)`, and `XCEGET` must be coded using message passing.
- Debugging is difficult. In particular, the correctness of a single tile code does not imply that the corresponding multi-tile code is correct. Data parallel codes can in principle be fully debugged on a single processor.
- Message passing is not efficient on a multi-user SMP system, although autotasking may reduce the importance of this fact.

2.3 Tiled Data Parallel

It is possibile to combine the tiled message passing and data parallel methods. Consider our simple code fragment:

```
REAL    A(IH+1,JH),DA(IH+1,JH),DX
INTEGER I,J
DO J= 1,JH
  DO I= 1,IH
    DA(I,J) = DX*(A(I+1,J) - A(I,J))
ENDDO; ENDDO
```

In tiled data parallel form, for CM Fortran, this becomes:

```
        REAL    DX
        REAL    A( 0:IHP+1,0:JHP+1,MP,NP)
        REAL    DA(0:IHP+1,0:JHP+1,MP,NP)
CMF$    LAYOUT  A( :SERIAL,:SERIAL,:NEWS,:NEWS)
CMF$    LAYOUT  DA(:SERIAL,:SERIAL,:NEWS,:NEWS)
        INTEGER I,J,M,N
        DO N= 1,NP
          DO M= 1,MP
            DO J= 1,JHP
              DO I= 1,IHP
                DA(I,J,M,N) = DX*(A(I+1,J,M,N) - A(I,J,M,N))
            ENDDO; ENDDO;
        ENDDO; ENDDO;
```

If `MP` and `NP` are both 1, this is functionally identical to the message passing code fragment. If `MPxNP` represents the number of processors (or, on the CM5, the number of vector units), this is data parallel and the compiler does not need to generate any off-chip communication. In principle, a tiled data parallel code should be as fast or faster than the corresponding native data parallel code, because it does less communication unless the compiler is caching off-chip values. However, the speed of a tiled data parallel code depends on how well the compiler handles serial, i.e. on-chip, operations and CM Fortran's serial optimizations are currently quite limited. Since CM Fortran requires array syntax, the CMAX preprocessor must be invoked on the CM5, as part of the compile phase, to convert the DO-loop nest to an array expression:

```
 DA(1:IHP,1:JHP,:,:) = DX*(A(2:IHP+1,1:JHP,:,:) -
&                          A(1:IHP,  1:JHP,:,:)  )
```

This is unnecessary in HPF, which will directly parallelize the DO loops.

The tiled data parallel programming style adds more boiler plate to the original code, to handle the tile number dimensions, but can use message passing to update halos etcetera and is in every way functionally equivalent to the standard tiled message passing style. It can also run on data parallel machines/compilers, using data parallel constructs in communication subroutines. Disadvantages include:

- Boiler plate reduces clarity, when compared to either native data parallel or tiled message passing codes.
- All non-local operations require a subroutine call, e.g. `AIJ = A(I,J)` becomes `CALL XCEGET(AIJ, A,I,J)`, and `XCEGET` must be coded using both data parallel and message passing.

On SMP machines autotasking can occur at the block level (outermost loop) even if the original code did not autotask, and the multi-tile code can be debugged on a single processor (except for the message passing communication routines).

The NLOM is now implemented in the tiled data parallel style, and this single code has replaced the original separate SMP and data parallel versions.

3. NLOM IMPLEMENTATION

3.1 Example of Autotasking Code

Consider a single subroutine, used as part of the pressure gradient calculation, from the original NLOM implementation.

```
      SUBROUTINE MHMVPO(DV, PTO,HHYTO, QTDTDY)
      IMPLICIT NONE
      INTEGER    IH,JH
      PARAMETER (IH=515, JH=287)
C
      REAL QTDTDY
      REAL DV(IH,JH),PTO(IH,JH),HHYTO(IH,JH)
C
      INTEGER        IFU,IFV,IFH,IFS,ILU,ILV,ILH,ILS
      COMMON/ZILOOP/ IFU(JH),IFV(JH),IFH(JH),IFS(JH),
     &               ILU(JH),ILV(JH),ILH(JH),ILS(JH)
      SAVE  /ZILOOP/
C
C     V MOMENTUM EQUATION - PRESSURE GRADIENT TERM.
C
      INTEGER I,J
C
      DO J= 2,JH-2
        DO I= IFV(J),ILV(J)
          DV(I,J) = DV(I,J) -
     &     QTDTDY*(PTO(I,J+1) - PTO(I,J))*HHYTO(I,J)
```

```
ENDDO; ENDDO
RETURN
END
```

The subroutine is acting on a single layer of a multi-layer model, so the subroutine would be called once for each layer with only horizontal slices of the full arrays passed via the argument list. All low level routines in the ocean model act this way. It allows workspace arrays, e.g. DV and HHYT0, to be two dimensional. A typical call to this subroutine might be:

```
CALL MHMVP0(DV, P(1,1,NT0,K),HHYT0, QTDTDY)
```

where `NT0` represents the time level and `K` represent the layer. This takes advantage of Fortran 77's ability to treat an array element as the starting point of an array slice. In Fortran 90, and CM Fortran, the slice would be made explicit:

```
CALL MHMVP0(DV, P(:,:,NT0,K),HHYT0, QTDTDY)
```

The subroutine is configured to vectorize the I-loop and autotask the J-loop. The inner loop does not cover the full array. Instead, it skips all calculations on `[1:IFV(J)-1,J]` and `[ILV(J)+1:IH,J]`, because these are all land points. This technique is known as *shrinkwrapping* [2]. Note that there may well be land points within `[IFV(J):ILV(J),J]`, and calculations are performed at such points. The boundary conditions imposed at the end of each time step will allow for land points, and therefore the presence of absence of shrinkwrapping does not affect correctness. The variable inner loop extent slows down vectorization, and causes some load inbalance when autotasking (i.e. it tends to reduce the substainable Mflop/s rate). However, it is almost always a net win in overall performance on SMP machines to shrinkwrap land, because the time saved by skipping calculations is larger than the time lost by less efficient vectorization and parallelization. The `/ZILOOP/ COMMON` contains four sets of shrinkwrap limits, for the four separate staggered grids used in the model calculation.

Some ocean models skip all land calculations, either by using a mask or by splitting the inner loop into many loops that are land-free. This allows them to run with simplified boundary corrections at the end of each time step, but they take a performance hit on every finite difference calculation (particularly on vector machines). NLOM typically spends less than 5% of its time in boundary routines, and this is easily made up for by increased finite difference code performance.

3.2 Example of Tiled Data Parallel Code

Portability often requires different code to be executed on different machine types. Since the code for machine X may not even be legal Fortran syntax on machine Y, the appropriate code fragment needs to be selected before the Fortran compiler sees the source code. The C preprocessor, cpp, is an integral part of standard C and, among other things, allows conditional compilation. There are alternatives, but cpp is commonly used as a Fortran preprocessor. In fact, many Fortran compilers follow the convention that files ending in .F or .F90 are first passed through the C preprocessor before entering the standard compilation phases. Macro subsitution, i.e. the replacement of one text string with another, is also a feature of cpp that can be useful in Fortran programs, although it sometimes produces counter-intuitive results. NLOM makes extensive use of cpp capabilities. The following are a set of global C preprocessor macros, that may be referenced by all NLOM subroutines.

```
/* C-preprocessor macros for customizing the ocean model for a computer */
/* Since f77 uses all UPPERCASE, macros use all lowercase               */

/* region size; rich in powers of 2 */
# define ih 512
# define jh 288

#if defined(spmd)
/* generic SPMD message passing version; ip=jp=1, no. proc = ipr*jpr */
# define ip     1
# define jp     1
# define ipr    4
# define jpr    4
#elif defined(hpf)
/* HPF data parallel version; no. proc = ip*jp */
# define ip     8
# define jp     4
#elif defined(cm5)
/* CM5 data parallel version; CMAX and CM Fortran; no. proc = ip*jp/4 */
# define ip    16
# define jp     8
#elif defined(thread)
/* Generic autotasked shared memory version; parallelize on jp */
# define ip     1
# define jp    16
# define shrinkwrap
#else
/* single node or shared memory version; parallelize on jh */
# define ip     1
# define jp     1
# define shrinkwrap
#endif

#if defined(ipr)
/* message passing tile size, IP=JP=1 */
# define ihp  IH/ipr
# define jhp  JH/jpr
#else
/* standard tile size */
# define ihp  IH/IP
# define jhp  JH/JP
#endif

#if defined(hpf)
/* HPF data parallel compiler directives */
# define layout_s2b2(a) !HPF$ DISTRIBUTE a(*,*,BLOCK,BLOCK)
#elif defined(cm5)
```

```
/* CM5 data parallel compiler directives */
# define layout_s2b2(a) CMF$  LAYOUT a(:SERIAL,:SERIAL,:NEWS,:NEWS)
#else
# define layout_s2b2(a)
#endif

#if defined(shrinkwrap)
/*  skip most calculations over land */
# define ilfm MLF(J,M,N)
# define ilf  ILF(J,M,N)
# define ill  ILL(J,M,N)
# define illp MLL(J,M,N)
#else
/*     do all calculations over land */
# define ilfm 1
# define ilf  2
# define ill  IHP+1
# define illp IHP+2
#endif
```

Now the tiled pressure gradient subroutine, with the above macro definitions invoked via `#include`:

```
#include "MACROS.cpp"
      SUBROUTINE MHMVPO(DV, PTO,HHYTO, QTDTDY)
      IMPLICIT NONE
      INTEGER    IH,JH,IP,JP,IHP,JHP
      PARAMETER (IH=ih, IP=ip, IHP=ihp)
      PARAMETER (JH=jh, JP=jp, JHP=jhp)
C
      REAL*4  QTDTDY
      REAL*4  DV( IHP+3,JHP+3,IP,JP),
     &        PTO(IHP+3,JHP+3,IP,JP),HHYTO(IHP+3,JHP+3,IP,JP)
layout_s2b2(DV)
layout_s2b2(PTO)
layout_s2b2(HHYTO)
#if defined(shrinkwrap)
      INTEGER        ILF,ILL,MLF,MLL
      COMMON/ZILOOP/ ILF(JHP+3,IP,JP),ILL(JHP+3,IP,JP),
     &               MLF(JHP+3,IP,JP),MLL(JHP+3,IP,JP)
      SAVE  /ZILOOP/
#endif
C
C     V MOMENTUM EQUATION - PRESSURE GRADIENT TERM.
C
      INTEGER I,J,M,N
C
      DO N= 1,JP
        DO M= 1,IP
          DO J= 2,JHP+1
```

```
        DO I= ilf,ill
          DV(I,J,M,N) = DV(I,J,M,N) -
&          QTDTDY*(PTO(I,J+1,M,N) - PTO(I,J,M,N))*
&          HHYTO(I,J,M,N)
      ENDDO; ENDDO;
 ENDDO; ENDDO;
 RETURN
 END
```

This is more complicated than the example used to introduce the tiled data parallel technique. The halo consists of one row/column on one side and two rows/columns on the other, because an extra row/column is required to handle some operations on the staggered grids. Arrays are dimensioned `(1:IHP+3,1:JHP+3,IP,JP)`, rather than `(0:IHP+2,0:JHP+2,IP,JP)`. The latter would simplify array indexing, but Fortran compilers are not consistant in how they handle arrays that don't start at `(1,1,1,1)`, and so the less transparent dimensioning scheme is used to enhance portability. The array size is slightly different than the original. `IH` no longer includes a halo to handle the global ocean periodic boundary (i.e. 512 rather than 515), and `JH` has been increased to a value rich in powers of two (288 vs 287, but much larger differences in size are possible). Array dimensions rich in powers of two allow the number of equally sized tiles to be a large power of two, which is optimal on some massively parallel machines. The non-standard data types `REAL*4` and `REAL*8` are used because they are the most portable way to get 32-bit `REAL` and 64-bit `DOUBLE PRECISION`. Data parallel compiler directives for array layout are included via macros, which allows both CM Fortran and HPF to be supported, and makes it possible to use different layouts on different machines. On SPMD machines, there is no advantage to shrinkwrapping because overall speed is controlled by the tile with the least land. When shrinkwrapping, "`ilf,ill`" is expanded to "`ILF(J,M,N),ILL(J,M,N)`" but otherwise it is "`2,IHP+1`". Tiling actually simplifies shrinkwrapping. All four staggered grids have the same extent, specified in `ILF` and `ILL`, but intermediate values may need to be calculated at halo points in which case `MLF` and/or `MLL` are used. Note that in the shrinkwrapped case, `MLF` and `MLL` are not necessarily `ILF-1` and `ILL+1` because, for example, a halo point can be a sea point even if the adjacent non-halo point is land.

The data parallel style requires all tiles to be the same size and there must be a one to one mapping between tiles and a 2-D mesh of processors. This can lead to some inefficiencies in handling land. On a SMP machine with relatively few processors, shrinkwrapped tiles don't all perform the same amount of work. Better load balance can be obtained by using variable sized tiles, with tile sizes chosen so that each tile contains about the same number of ocean points. NLOM reduces SMP load inbalance by the alternative technique of having more tiles than processors, e.g. 12 tiles for 2, or 3, or 4 processors. On a large message passing machine, some tiles will contain only land and hence do no useful work. There is no need to include these empty tiles in the configuration at all. This is relatively simple to implement, since the bulk of the model code does not know about other tiles in any case. A tiled data parallel code in data parallel mode would have to include all tiles, but some could still be missing when running the same code with message passing. NLOM does not use this optimization, because the Capacitance Matrix Technique it uses to solve Helmholtz's equations performs most calculations at all points and because parallel I/O is more difficult with missing tiles. Also, some massively parallel machines have explicit support for, and run optimally over, a 2-D grid of processors. On the other hand, most machines can handle missing tiles with little communication overhead and the potential saving in time is significant (20% to 40% of points are typically over land). NLOM may eventually be extended to skip land tiles, and this is certainly

an optimization worth considering for any message passing ocean model. The scalable version of the MICOM model [2], includes both the variable tile size and empty tile optimizations.

3.3 Communication Subroutines

The following are one line descriptions of all NLOM 2-D array communication routines:

```
XCAGET - convert an array from tiled to non-tiled layout
XCAPUT - convert an array from non-tiled to tiled layout
XCBGET - extract A(IA:IA+1,JA:JA+1) from a tiled array
XCEGET - extract the element A(IA,JA) from a tiled array
XCEWGT - extract the element A(IA,JA) from a tiled wind array
XCHALT - emergency stop all processes, called by one process
XCLGET - extract a line from a tiled array
XCLGT2 - extract a double line from a tiled array
XCLPUT - place a line of values into a tiled array
XCMASS - sum a tiled array
XCMAS1 - sum a sub-region of a tiled array
XCMINH - find the minimum value in a tiled array
XCNORM - find the L2 norm of a tiled array
XCRANG - find value and location of min and max from tiled array
XCSHFT - periodic shift of a tiled array
XCSPMD - initialize processor data structures, called once
XCSTOP - stop all processes, called by all processes
XCTILB - update a tiled array's halo
XCTILI - initialize for halo communication, called once
XCTILR - update a tiled array's halo, for red-black SOR
XCTILV - partial update of U,V tiled array's halos
XCTILW - update a tiled wind array's halo
```

With the exception of `XCHALT`, all these routines are assumed to be called with identical argument lists by all processors when using SPMD message passing. This is not difficult to arrange, since by default all routines are called in this manner in a SPMD run. Most communication routines act as implicit barriers that synchronize processor state, i.e. when a processor exits a communication routine at the very least all processors that must communicate with it have entered the same subroutine. In addition the macro "`barrier`" is provided for cases where all processors must enter a critical section of code before the first processor exits. This is implemented as a macro, rather than a subroutine, to reduce overhead for autotasking and data parallel cases which never need user invoked barriers (since the compiler synchronizes the processors, whenever necessary).

NLOM is designed to produce identical results on a given system no matter how many processors are used. This greatly improves robustness and simplifies debugging. Global sum routines (`XCMASS` and `XCNORM`), cannot use the fast algorithms typically implemented by data parallel compilers and by message passing libraries because these are not independant of the number of processors involved in the sum. Instead, a pipelining implementation is used. Pipelining is described in the context of a tridiagonal solver in section 3.7.

Three versions of each communication subroutine are provided; one for message passing, one for autotasking/data parallel using Fortran 77 syntax, and one for data parallel using Fortran 90

syntax. The routine XCTILB will be used as an example. It updates the halo of a standard tiled array. The Fortran 90 data parallel version just uses CSHIFT on a halo-only buffer:

```
      SUBROUTINE XCTILB(A)
      IMPLICIT NONE
      INTEGER    IH,JH
      PARAMETER (IH=ih, JH=jh)
C
      INTEGER    IP,JP,IHP,JHP,IHP3,JHP3
      PARAMETER (IP  =ip,   JP  =jp)
      PARAMETER (IHP =ihp,  JHP =jhp)
      PARAMETER (IHP3=ihp3, JHP3=jhp3)
C
      REAL*4  A(IHP3,JHP3,IP,JP)
layout_s2b2(A)
C
C**********
C*
C 1)  UPDATE TILE OVERLAP.
C*
C**********
C
      INTRINSIC CSHIFT
C
      REAL*4  AB1( IHP3,2,IP,JP),AB2( IHP3,1,IP,JP),
     &        AB3( 2,JHP3,IP,JP),AB4( 1,JHP3,IP,JP)
      REAL*4  AB1S(IHP3,2,IP,JP),AB2S(IHP3,1,IP,JP),
     &        AB3S(2,JHP3,IP,JP),AB4S(1,JHP3,IP,JP)
layout_s2b2(AB1)
layout_s2b2(AB2)
layout_s2b2(AB3)
layout_s2b2(AB4)
layout_s2b2(AB1S)
layout_s2b2(AB2S)
layout_s2b2(AB3S)
layout_s2b2(AB4S)
C
      AB2  = A(:,JHP+1:JHP+1,:,:)
      AB1  = A(:,    2:3,    :,:)
      AB2S = CSHIFT(AB2,SHIFT=-1,DIM=4)
      AB1S = CSHIFT(AB1,SHIFT=+1,DIM=4)
      A(:,    1:1    ,:,:) = AB2S
      A(:,JHP+2:JHP+3,:,:) = AB1S
C
      AB4  = A(IHP+1:IHP+1,:,:,:)
      AB3  = A(    2:3,    :,:,:)
      AB4S = CSHIFT(AB4,SHIFT=-1,DIM=3)
      AB3S = CSHIFT(AB3,SHIFT=+1,DIM=3)
```

```
      A(    1:1    ,:,:,:) = AB4S
      A(IHP+2:IHP+3,:,:,:) = AB3S
      RETURN
C     END OF XCTILB.
      END
```

All versions of XCTILB are doubly periodic, which simplifies programming logic. In cases with closed boundaries, the periodic wrapped halos will contain land points, and therefore do no harm. The autotasking XCTILB uses indirect addressing to get to the tile it requires:

```
      SUBROUTINE XCTILB(A)
      IMPLICIT NONE
      INTEGER    IH,JH
      PARAMETER (IH=ih, JH=jh)
C
      INTEGER    IP,JP,IHP,JHP,IHP3,JHP3
      PARAMETER (IP  =ip,   JP  =jp)
      PARAMETER (IHP =ihp,  JHP =jhp)
      PARAMETER (IHP3=ihp3, JHP3=jhp3)
C
      REAL*4  A(IHP3,JHP3,IP,JP)
layout_s2b2(A)
C
      INTEGER          MM,MP,NM,NP
      COMMON/ZCTILE/   MM(IP,JP),
     &                 MP(IP,JP),
     &                 NM(IP,JP),
     &                 NP(IP,JP)
layout_b2(MM)
layout_b2(MP)
layout_b2(NM)
layout_b2(NP)
      SAVE  /ZCTILE/
C
C**********
C*
C  1) UPDATE TILE OVERLAP.
C
C  2) /ZCTILE/ MUST FIRST BE SET BY CALLING XCTILI.
C*
C**********
C
      INTEGER I,J,M,N
C
      DO 010 N= 1,JP
        DO 012 M= 1,IP
          DO 110 I= 2,IHP+1
            A(I,    1,M,N) = A(I,JHP+1,M,NM(M,N))
            A(I,JHP+2,M,N) = A(I,    2,M,NP(M,N))
```

```
            A(I,JHP+3,M,N) = A(I,     3,M,NP(M,N))
  110     CONTINUE
  012   CONTINUE
  010 CONTINUE
C
      DO 020 N= 1,JP
        DO 022 M= 1,IP
          DO 210 J= 1,JHP+3
            A(    1,J,M,N) = A(IHP+1,J,MM(M,N),N)
            A(IHP+2,J,M,N) = A(    2,J,MP(M,N),N)
            A(IHP+3,J,M,N) = A(    3,J,MP(M,N),N)
  210     CONTINUE
  022   CONTINUE
  020 CONTINUE
      RETURN
C     END OF XCTILB.
      END
```

Due to the periodic wrap, `MM(1,:)=IP`, `MP(IP,:)=1`, `NM(:,1)=JP`, and `NP(:.JP)=1`. Finally, here is the message passing version.

```
      SUBROUTINE XCTILB(A)
      IMPLICIT NONE
      INTEGER    IH,JH
      PARAMETER (IH=ih, JH=jh)
C
      INTEGER    IP,JP,IPR,JPR,IHP,JHP,IHP3,JHP3
      PARAMETER (IP  =ip,   JP  =jp)
      PARAMETER (IPR =ipr,  JPR =jpr)
      PARAMETER (IHP =ihp,  JHP =jhp)
      PARAMETER (IHP3=ihp3, JHP3=jhp3)
C
      REAL*4  A(IHP3,JHP3,IP,JP)
C
      INTEGER          MPROC,NPROC
      COMMON/CPROCI/   MPROC,NPROC
      SAVE  /CPROCI/
C
      INTEGER          IDPROC
      COMMON/CPROCD/   IDPROC(0:IPR+1,0:JPR+1)
      SAVE  /CPROCD/
C
C**********
C*
C 1)  UPDATE TILE OVERLAP.
C*
C**********
C
      INTEGER    ILEN,JLEN
```

```
      PARAMETER (ILEN=IHP*IP, JLEN=(JHP+4)*JP)
C
C     MAKE BUFFER SIZES A MULTIPLE OF 2.
C
      REAL*4  AI(IHP,IP,6),AJ(JHP+4,JP,6)
      SAVE    AI,AJ
C
      INTEGER I,J,M,N
C
      header
C
      IF     (JP.NE.1) THEN
        DO 110 N= 1,JP-1
          DO 111 M= 1,IP
            DO 112 I= 2,IHP+1
              A(I,    1,M,N+1) = A(I,JHP+1,M,N)
              A(I,JHP+2,M,N)   = A(I,    2,M,N+1)
              A(I,JHP+3,M,N)   = A(I,    3,M,N+1)
  112       CONTINUE
  111     CONTINUE
  110   CONTINUE
      ENDIF
      IF     (JPR.EQ.1) THEN
        DO 120 M= 1,IP
          DO 121 I= 1,IHP
            A(I+1,    1,M,1)  = A(I+1,JHP+1,M,JP)
            A(I+1,JHP+2,M,JP) = A(I+1,    2,M,1)
            A(I+1,JHP+3,M,JP) = A(I+1,    3,M,1)
  121     CONTINUE
  120   CONTINUE
      ELSE
        DO 130 M= 1,IP
          DO 131 I= 1,IHP
            AI(I,M,1) = A(I+1,JHP+1,M,JP)
            AI(I,M,2) = A(I+1,    2,M,1)
            AI(I,M,3) = A(I+1,    3,M,1)
  131     CONTINUE
  130   CONTINUE
#if defined(mpi)
        CALL MPI_SENDRECV(
     +          AI(1,1,1),  ILEN,mtyper,IDPROC(MPROC,NPROC+1), 9905,
     +          AI(1,1,4),  ILEN,mtyper,IDPROC(MPROC,NPROC-1), 9905,
     +          MPI_COMM_WORLD, MPISTAT, MPIERR)
        CALL MPI_SENDRECV(
     +          AI(1,1,2),2*ILEN,mtyper,IDPROC(MPROC,NPROC-1), 9906,
     +          AI(1,1,5),2*ILEN,mtyper,IDPROC(MPROC,NPROC+1), 9906,
     +          MPI_COMM_WORLD, MPISTAT, MPIERR)
```

```
#elif defined(shm)
        barrier
        CALL shmem_getr(AI(1,4),
     +                  AI(1,1),  ILEN,  IDPROC(MPROC,NPROC-1))
        CALL shmem_getr(AI(1,5),
     +                  AI(1,2),2*ILEN,  IDPROC(MPROC,NPROC+1))
#endif
        DO 140 M= 1,IP
          DO 141 I= 1,IHP
            A(I+1,    1,M,1)  = AI(I,M,4)
            A(I+1,JHP+2,M,JP) = AI(I,M,5)
            A(I+1,JHP+3,M,JP) = AI(I,M,6)
  141     CONTINUE
  140   CONTINUE
      ENDIF
C
      IF     (IP.NE.1) THEN
        DO 210 N= 1,JP
          DO 211 M= 1,IP-1
            DO 212 J= 1,JHP+3
              A(    1,J,M+1,N) = A(IHP+1,J,M,  N)
              A(IHP+2,J,M,  N) = A(    2,J,M+1,N)
              A(IHP+3,J,M,  N) = A(    3,J,M+1,N)
  212       CONTINUE
  211     CONTINUE
  210   CONTINUE
      ENDIF
      IF     (IPR.EQ.1) THEN
        DO 220 N= 1,JP
          DO 221 J= 1,JHP+3
            A(    1,J,1, N) = A(IHP+1,J,IP,N)
            A(IHP+2,J,IP,N) = A(    2,J,1, N)
            A(IHP+3,J,IP,N) = A(    3,J,1, N)
  221     CONTINUE
  220   CONTINUE
      ELSE
        DO 230 N= 1,JP
          DO 231 J= 1,JHP+3
            AJ(J,N,1) = A(IHP+1,J,IP,N)
            AJ(J,N,2) = A(    2,J,1, N)
            AJ(J,N,3) = A(    3,J,1, N)
  231     CONTINUE
  230   CONTINUE
#if defined(mpi)
        CALL MPI_SENDRECV(
     +        AJ(1,1,1),  JLEN,mtyper,IDPROC(MPROC+1,NPROC), 9907,
     +        AJ(1,1,4),  JLEN,mtyper,IDPROC(MPROC-1,NPROC), 9907,
```

```
     +        MPI_COMM_WORLD, MPISTAT, MPIERR)
         CALL MPI_SENDRECV(
     +        AJ(1,1,2),2*JLEN,mtyper,IDPROC(MPROC-1,NPROC), 9908,
     +        AJ(1,1,5),2*JLEN,mtyper,IDPROC(MPROC+1,NPROC), 9908,
     +        MPI_COMM_WORLD, MPISTAT, MPIERR)
#elif defined(shm)
         barrier
         CALL shmem_getr(AJ(1,1,4),
     +                   AJ(1,1,1),  JLEN,IDPROC(MPROC-1,NPROC))
         CALL shmem_getr(AJ(1,1,5),
     +                   AJ(1,1,2),2*JLEN,IDPROC(MPROC+1,NPROC))
#endif
         DO 240 N= 1,JP
           DO 241 J= 1,JHP+3
             A(    1,J,1, N) = AJ(J,N,4)
             A(IHP+2,J,IP,N) = AJ(J,N,5)
             A(IHP+3,J,IP,N) = AJ(J,N,6)
  241      CONTINUE
  240    CONTINUE
       ENDIF
       RETURN
C      END OF XCTILB.
       END
```

This version is longer than the others, because it allows multiple tiles per process and therefore includes the autotasking version's logic as well as message passing logic. Multiple tiles per process might be used on a SMP cluster, for example, with each process on a separate SMP machine communicating with other processes on other machines via message passing but parallelizing across multiple processors internally via autotasking. We will follow the usual convention of treating process and processor as interchangable terms to mean a single message passing entity, even though such an entity may in fact be muliple autotasked processors. Message passing uses either the T3D's SHMEM library or the standard MPI library [13]. Several macros are used to enhance portability. The macro "`header`" allows for a machine specific include file, "`mtyper`" allows for machine specific REAL data type, "`barrier`" blocks until all processors execute it, and macro "`shmem_getr`" allows for different subroutine names in T3D and SGI versions of the library. The variables `MPROC` and `NPROC` identify the local processor, with respect to a 2-D mesh or processors, and the array `IDPROC` contains the mapping from the mesh to processor number. `IDPROC` includes a halo to simplify periodic shifts. So, `IDPROC(MPROC,NPROC)` is the processor number of this processor, and its northern neighbour is processor number `IDPROC(MPROC,NPROC+1)`, etcetera. The processor array is configured to periodic wrap in both dimensions. On small workstation clusters it might be more efficient to skip the wrap when it isn't needed. SHMEM implements one sided communication, and can get values in any static array from another processor, the `SAVE` statement makes `AI` and `AJ` static. NLOM requires `IHP` and `JHP` to be even, so the first dimensions of `AI` and `AJ` are also even. This avoids performance problems that occur when the local and remote buffers are misaligned in memory. The barrier before each `shmem_getr` call ensures that the buffer, `AI` or `AJ`, is up to date on the remote processor before it is copied. In general, a barrier would also be required after the copies to ensure the remote memory is not altered before it is copied. This is not required here, because a total of two barriers are sufficient to protect two buffers (either a classic double buffer

or, as here, two independent buffers called in sequence). SHMEM is a particularly good match to the tiled data parallel programming style, because global barriers can always be placed between each stage of a data parallel algorithm allowing safe access to remote memory. The MPI version is not necessarily optimal on a given machine, but it is a portable starting point for a machine-specific MPI implementation. An optimized version might not use the buffers `AI` and `AJ`, which are essential for SHMEM but represent potentialy unnecessary copy operations under MPI. The subroutine `MPI_SENDRECV` is a simple way to implement a circular shift. It sends one set of values to one processor while receiving another set from a second processor. Since both ends cooperate in passing messages, there is no need for a global barrier. The default SPMD global communicator `MPI_COMM_WORLD` is being used here. On some machines a communicator based explicitly on a 2-D mesh might be more efficient.

3.4 Parallel I/O

NLOM is used on many machine types, so it is important that NLOM files be portable. This is acomplished, in part, by using IEEE 754 32-bit floating point values for all I/O. Integer values are converted to floating point, because integers can be either 32-bit or 64-bit and are hence non-portable. To allow some degree of parallel I/O on distributed memory machines, separate files are used for scalars and arrays. Scalar files are formatted or unformatted sequential, and array files are Fortran direct access with one 2-D array (representing a single field for a single layer) per record. Array I/O is performed via subroutine calls, which can do I/O in parallel provided the resulting file is configured as required. On the Cray C90, an assign statement is used (actually an ASNUNIT subroutine call) to configure for IEEE I/O. On DEC machines, the f77 compiler switch "`-convert big_endian`" is used to make the files compatable with the majority of Unix systems (DEC is little endian, but most others are big endian by default).

Our implementation of I/O is somewhat unusual, but a natural one for message passing SPMD codes. All files are opened either `READONLY` or `WRITEONLY`. Scalar files are read independently by all processors, but written by the first processor only. All processors can write to `STDOUT`, Fortran unit 6, but reading from `STDIN`, Fortran unit 5, is not allowed because on some machines it is not available to all processors. Fortran units 0 and 7 are not used, because they can have default bindings, to `STDERR`, which are not portable. Under a data parallel or autotasking compiler, scalar code and scalar I/O do not see multiple processors and so all I/O is from the single program instance. How array files are handled depends on the machine. Writing the entire array from the first processor using direct access I/O always works, and is the only choice for single node and autotasked cases. However, on distributed memory machines it requires all tiles to be message passed to one processor. An alternative is to use direct access I/O from the first processor in each row, reading/writing the entire rows's set of tiles as a single record. Each active processor is then accessing a different, non-overlapping, record in the file. This is always safe for parallel reading and is usually safe for parallel writing if the record length is a multiple of the disk block size. On a data parallel machine, copying from a tiled (4-D) array to a native (2-D) array and then using data parallel I/O is an option. In general, any method that reads/writes the standard file is allowed. There are several efforts underway to produce standards for parallel I/O, but it is not yet clear if any of them will be compatable with serial I/O. The NLOM technique maintains portability, but does not necessarily allow maximum parallel I/O performance. This is acceptable, because NLOM spends much more time computing than doing I/O. Using two files, scalar and array, to hold values is less robust than mixing them in the same file. We must rely on file name association to tell us

which scalar file goes with which array file, but it is not generally possible to do portable parallel I/O on files than mix scalars and arrays. Array files contain all land points. They can be reduced in size, after the fact, either by standard file compression techniques, e.g. via the gzip command, or run length encoding of land points via the NLOM-specific command lzip, or by converting them to the original NLOM history file format (a 24-bit fixed point value file containing no land values at all).

3.5 Serial Optimizations

The original implementation of NLOM was optimized for multi-processor vector machines, such as the Cray C90. Its performance on RISC microprocessor based workstations and SMP servers was of less importance. RISC-based systems now represent a much more significant fraction of the high performance computing field. So, the scalable NLOM implementation must be nearly optimal over a wider range of machines, including both vector and RISC processors. Vector machines can be characterized as having no cache and a very high memory to processor bandwidth. Arrays are often used to hold temporaries, and a non-unit stride through an array can be efficient unless the stride is a large power of two (e.g. 64 or 128). RISC-based machines always include a cache, often a multi-level cache, and have a low memory to cache bandwidth. Non-unit strides through an array can be expensive, unless the required elements are already in cache.

With the exception of the Helmholtz's equation solvers, described in section 3.7, the original NLOM code structure was already sufficiently clean to optimize well on RISC systems. There were no unnecessary non-unit strides through arrays, and the loop nests were simple enough for the compiler to automatically unroll most inner loops. Loop unrolling is essential for good performance on a RISC machine, and a few loops had to be unrolled by hand, with the original code retained for vector machines via cpp logic, when one or more compilers failed to recognize the possibility of unrolling. A few array temporaries have been replaced by scalars, but the original NLOM code typically used array temporaries, rather than scalars, only when they reduced the operation count. So most array temporaries have been retained. Using 32-bit `REAL`'s effectively doubles the cache size and the memory to cache bandwidth. This provides a performance boost even on machines that run at the same speed for 32-bit and 64-bit floating point arithmetic. All ocean models should be able to use 32-bit IEEE 754 `REAL`s for most calculations.

Division is always expensive, but it is particularly costly on some RISC systems. NLOM already performed the minimum possible number of divisions. The IEEE 754 floating point arithmetic standard does not allow division to be replaced by multiplication by a reciprocal. A compiler that strictly enforces this requirement could not replace:

```
DO I= 1,N
  A(I) = B(I)/C
ENDOD
```

with:

```
RC = 1.0/C
DO I= 1,N
  A(I) = B(I)*RC
ENDOD
```

even though the latter might be ten times faster. This is one of the few situations where it is still advantageous for a programmer to promote loop independent calculations to outside the loop.

Codes optimized for vector, or data parallel, machines tend to make liberal use of simple vector operations, such as `Y=X`, `Y=0.0`, and `Y=Y+X`. These are relatively inexpensive on vector machines, but can take a significant amount of time on machines with a low memory bandwidth. The obvious first step is to remove unnecessary operations of this sort. However, good programming practice and numerical accuracy considerations have lead to many such operations being retained in NLOM. So they have optionally been replaced by calls to optimized BLAS, Basic Linear Algebra Subroutines [10], where available. The BLAS were designed only to support the needs of linear algebra, and not all the required vector operations are included. Moreover, many vendors do not optimize the original level-1 (vector-vector) BLAS, because vector lengths are short in linear algebra. Level-2 (matrix-vector) BLAS [4] and level-3 (matrix-matrix) BLAS [5] have more room for optimization and are more important to modern linear algebra applications. There is a need for a standardized vector library to facilitate the porting of codes from vector to RISC architectures. This could be implemented in assembly language, if necessary, for each RISC architecture. For ocean models, a level-2 BLAS extension, subroutine `SGESUMC`, implementing the matrix (2-D array) operation "B = alpha*op(A) + beta*B + gamma", where op(A) is either A or its transpose, would be sufficient to cover most simple vector operations (provided cases where the scalars alpha, beta, and gamma are zero or one are treated optimally). Cray has a similar BLAS extension, `SGESUM`, on the T3D only. To illustrate that Fortran compilers do not produce optimal code for simple vector operations, consider two implementations of the same subroutine. First, the obvious Fortran version.

```
      SUBROUTINE R4WSET(S,W,N)
      IMPLICIT NONE
      INTEGER N
      REAL*4  S(N),W
C
C     S = W.
C
      INTEGER I
C
      DO I= 1,N
        S(I) = W
      ENDDO
      RETURN
      END
```

Then, a version that is often faster on RISC systems.

```
      SUBROUTINE R4WSET(S,W,N)
      IMPLICIT NONE
      INTEGER N
      REAL*4  S(N),W
C
C     S = W.
C
      INTEGER LOC
      INTEGER IS1
```

```
      REAL*8  W8(1)
      REAL*4  W4(2)
      EQUIVALENCE (W8,W4)
C
      W4(1) = W
      W4(2) = W
      IS1   = LOC(S(1))
      IF     (MOD(IS1,8).EQ.0) THEN
        CALL R8WSET(S(1),W8,N/2)
        S(N) = W
      ELSE
        S(1) = W
        CALL R8WSET(S(2),W8,(N-1)/2)
        S(N) = W
      ENDIF
      RETURN
      END
```

The second version uses the non-standard `LOC` function to return the address of `S(1)` in bytes, and hence to select `S(1)` or `S(2)` as a legal `REAL*8` pointer. It then calls a `REAL*8` version of the subroutine with a `REAL*8` constant containing two copies of `W` and about half the original vector length. This version is not standard Fortran 77, but it is significantly faster than the original on many machines because the memory bandwidth is higher for 8-byte memory writes than for 4-byte writes. Presumably, an assembly language version of `R4WSET` could be faster still. The scalable NLOM code includes the option of using optimized BLAS and 64-bit vs 32-bit optimizations when appropriate, but a standard vector library optimized for each machine would simplify NLOM logic and further improve performance.

3.6 Cache Management

All ocean models are intrinsically memory intensive. The ratio of floating point operations to memory accesses is small, and all array elements are accessed at least once every one or two time steps. The previous section discussed essentially local optimizations. This section is concerned with global code restructuring for better cache performance. On a SGI Power Challenge, with a 4 MB secondary cache, a very small NLOM test case runs at 103 Mflops/s but a more realisticly sized case runs at only 39 Mflop/s. The two cases were not exactly comparable, but since the small case is running almost entirely cache contained it does place an upper bound on what cache optimizations can do. The NLOM code slabs on layers, i.e. all lower level routines deal with a single layer and the loop over all layers is outside these routines. Moreover, each low level routine implements a relatively small and self contained operation, for example there are separate routines for advection and diffusion, and there are several involved in calculating pressure gradients. This programming style is flexible and easy to maintain, but it does require more passes though each set of layer arrays than is strictly necessary. Layer slabs are large and don't typically fit in cache, so low level single layer subroutines are not optimal for cache management. A monolithic programming style where all the finite difference operations for a time step were performed by one big loop (or a small number of loops), would maximize locality of reference, but compilers don't always perform well when optimizing very complicated loops and large loops are difficult to maintain.

In principle, the message passing version of NLOM should already be configurable for optimal cache management. All that is required is to set the tile size so that a complete set of arrays for a single layer fit in cache. This would typically result in more tiles, and therefore processes, than physical processors. If these processes could be optimally scheduled by the operating system and if the per process message passing overhead were small, we would have a near optimal configuration with good cache performance. However, most message passing libraries assume there is one dedicated processor for each process and Unix time slices all processes and schedules them in a round robin (which is far from optimal). Optimal scheduling would consist of each process continuing to run on a physical processor until it blocks, at a barrier or a message passing operation, at which point it sleeps and another process takes over until it blocks, etcetera. A similar scheduling scheme is available, and works well, for some thread libraries [11], but schemes that work for light weight threads may be impractical for heavy weight processes. The direct use of threads, with one thread per tile, would allow better scheduling and might in fact lead to the fastest possible implementation on those SMP machines that support the mapping of many threads onto fewer processors. However, threads are not a natural fit to the SPMD programming style, and a combination of threads and message passing would be required when running across a cluster of SMP systems. Autotasking makes use of threads on many machines and, even if autotasking uses heavy weight processes, the master-slave algorthm used to parallelize DO-loops provides near optimal scheduling. What is required is to move the tile loops out of low level subroutines, and call as many subroutines as possible within each autotasked tile loop. However tile loops cannot contain calls to communication routines. In pseudo-code, for `IP=1`, this might look like:

```
DO N= 1,JP
  ! LOW LEVEL ROUTINES HANDLE A SINGLE TILE, HENCE (1,1,1,N).
  CALL SUB1(DU(1,1,1,N), U(1,1,1,N,NTO,K), .... )
  CALL SUB2(DV(1,1,1,N), H(1,1,1,N,NTO,K), .... )
  CALL SUB3(VV(1,1,1,N), V(1,1,1,N,NTO,K), .... )
  CALL SUB4(UV(1,1,1,N), U(1,1,1,N,NTO,K), .... )
ENDDO
CALL XCTILB(VV) ! COMMUNICATION ROUTINES HANDLE ENTIRE ARRAYS
CALL XCTILB(UV) ! AND STILL CONTAIN INTERNAL DO N= 1,JP LOOP
DO N= 1,JP
  CALL SUB5(DV(1,1,1,N), VV(1,1,1,N),V(1,1,1,N,NTO,K), .... )
  CALL SUB6(DV(1,1,1,N), VV(1,1,1,N),H(1,1,1,N,NTO,K), .... )
ENDDO
```

As it stands, this is not data parallel unless `SUB1` to `SUB6` are treated as `HPF EXTRINSIC` subroutines and these, while part of the standard language, are unlikely to be portable. However, compatability with data parallel can be maintained by using cpp macros to configure the same code to put tile loops either inside or outside low level routines. Compiler directives would typically be needed to force autotasking on these `N` loops, because the compiler would otherwise have to assume the subroutine calls were not thread safe.

Tile loops outside low level routines can also improve cache performance on a single processor. So a generic advantage of tiled data parallel over tiled message passing is that each clustered system can run a single program that itself handles multiple tiles, either to improve cache performance on a single processor, or to use multiple shared memory processors via autotasking. Hybrid parallelization techniques, such as autotasking plus message passing, are likely to be increasing important in the future, due to the growing popularity of SMP clusters.

The NLOM implementation does not currently support tile loops outside low level subroutines. A test case that simulated such loops without the synchronization required for communication routines, i.e. a best case test, ran 35% faster on a SGI Power Challange, but only 19% faster on a Sun SPARCstation 20, compared to the standard scalable code. A test case that was entirely cache contained but performed equal work, ran 55% and 29% faster than the standard case on SGI and Sun machines respectively. So there is some overhead whenever the model is larger than the cache, but a version with high level autotasked loops might perform significantly better than autotasking at the lowest loop nest level. The difference in speedup on the Sun and SGI is due to the relatively better balance between memory and cache bandwidth on the Sun. The existing generation of multi-processor servers, like the SGI Power Challange, typically use a shared memory bus that does not scale well as more processors are added. The next generation of servers are more likely to use memory crossbar switches, which have better bandwidth and scalability properties, and should therefore have less need for agressive cache optimization.

3.7 Parallelizing the Capacitance Matrix Technique

The NLOM gets its intrinsic efficiency from the use of Lagrangian layers in the vertical and a semi-implicit treatment of gravity waves. A semi-implicit model treats a linearized pressure gradient implicitly within an otherwise explict model. The time step is then independent of gravity wave speed, but the continuity equation becomes 3-D elliptic. The NLOM is designed so that the 3-D elliptic equation can be transformed to modal space, where it becomes a decoupled set of 2-D Helmholtz's equations (one per mode) [15]. The internal modes equations can each be solved with 4 to 10 iterations of Red-Black SOR, a classical iterative method with good scalabilty properties, but the single external mode equation is less well conditioned and is solved by the Capacitance Matrix Technique, CMT. Since explicit finite difference code is highly scalable, the overall scalability of NLOM depends on how scalable the CMT solver is. The fully scalable alternative to a semi-implicit time step is a split-explicit scheme that uses a smaller time step for the external gravity mode only. The split-explicit scheme is not competitive with semi-implicit on a single processor, because of its much higher operation count, but its superior scalability makes it relatively more efficient on a large number of processors. Thus our initial design goal was to make the CMT sufficiently scalable so that the semi-implicit model would out perform an equivalent split-explicit version. This goal has been exceeded, and NLOM is certainly still the most efficient existing basin-scale ocean model on any machine with up to hundreds of processors or even, if the problem is large enough, a few thousand processors.

The CMT [3], is a method for extending fast direct solvers for Helmholtz's equation over a separable region, such as a rectangle, to arbitrary bounded regions. Perhaps the simplest fast direct method is now usually called FACR(0) [6]. For an `IH` by `JH` rectangle, it involves `JH` independent forward FFT's each of length `IH`, followed by `IH` independent solves of tridiagonal systems of length `JH`, and finally `JH` independent reverse FFT's of length `IH`. The CMT involves a FACR(0) solve, followed by a boundary correction via the solution of a dense set of linear equations at boundary points, and finally a FACR(0) solve to apply the correction to the entire region. The Capacitance matrix contains every boundary node that is inside the rectangle, and so it can be very large. For a 1/32nd degree global region it is about an 100,000 by 100,000 matrix. However the matrix is not time dependent, so it can be inverted once per region and then one matrix-vector multiply used to solve the system every time step. The memory required for the Capacitance matrix is large, but its per time step cost is low and highly scalable. Thus the scalability of CMT depends on the

scalability of FACR(0).

There are two basic classes of solution methods for any operation that requires non-local communications. In distributed methods, the operation is performed in place with any non-local communication forming part of the solution process. In transpose methods, the layout of the array across processors is rearranged so that the operation itself requries no additional communication. Here transpose is used in a more general sense than normal, although on a SMP system the most common form of array layout rearrangement is the classical matrix transpose. Distributed methods typically involve passing short messages, with the possibility of overlapping computation with communication. Transpose methods typically involve passing long messages during a communication phase, which is separate from the computation phase. On the CM5, NLOM uses calls to the vendor provided CMSSL scientific library to perform FFT's and tridiagonal solves. These implement a distributed solution method, but the NLOM arrays must be converted, i.e. transposed, from tiled to non-tiled form before calling the CMSSL routines and then converted back to tiled form afterwards. Most scalable system vendors do not provide their own parallel, i.e. distributed, scientific library. So NLOM generically implements FACR(0) by first transposing from a 2-D processor grid to a 1-D grid, such that each of the independent FFT's are resident on one processor. This allows an optimized single processor FFT routine to be used. The tridiagonal solve could be implemented by doing a second transpose to a 1-D grid on the other array dimension, but NLOM instead uses a distributed algorithm to save the cost of the transposes. After the tridiagonal solve, the array is still in the 1-D layout that is optimal for the reverse FFT's. A FACR(0) implementation would normally be completed by a transpose from 1-D back to the standard 2-D array layout, but as part of the CMT implementation the reverse transpose after the first FACR(0) and the forward transpose before the second FACR(0) are not necessary.

The tridiagonal solver needs to solve `IH` independent systems each of which is `JH` long and distributed across all `P=NP*MP` processors. For simplicitly, NLOM normally requires that `JH` be a multiple of `P`. Using the "burn at both ends" (BABE) solution method doubles the available parallelism, and a data parallel implementation of pipelining is used as the primary distributed algorithm. Pipelining works by passing the results from solving a subset of the systems onto the next tile before continuing with the rest. In the first pass, start by solving the first `L` systems on processors `1` and `P` with processors `2` to `P-1` idle. In the second pass, processors `2` and `P-1` can work on the first `L` systems (using the results of the first pass) while processors `1` and `P` work on the second `L` systems with processors `3` to `P-2` idle. The subsequent passes are similar, but eventually processors (starting with `1` and `P`) run out of work, and go idle waiting for the rest of the processors to complete. Overall, there are `IH/L+P/2-1` passes and each processor is idle on `P/2-1` of them. Parallelization overhead is therefore theoretically minimized when `L=1`, but this is not optimal from a vectorization or message passing overhead standpoint. The optimal `L` can be determined experimentally, it is typically between 16 and 64 with higher values being appropriate for vector machines and for message passing over high latency networks. An important property of pipelining is that each tridiagonal system is solved using exactly the same operations, in exactly the same order, no matter how many processors, or tiles, are involved. This contributes to the overall invariance of the portable CMT implementation to the number of processors used. Maintaining processor independance leads to a much more robust code that is easier to debug. This is one reason for avoiding vendor provided distributed routines. In fact, the CM5 version of NLOM does not maintain exact processor independance because it uses CMSSL FFT and tridiagonal solver routines and CM Fortran array sum intrinsics. In this case exact invariance has been given up to

obtain acceptable performance.

The portable scalable CMT solver includes two different Fortran 77 FFT implementations, one primarily for vector machines and the other for RISC processors. On many systems a vendor provided FFT routine would be used instead. The primary disadvantage of transposing to a 1-D tiling is that it requires the second array dimension, JH, to be an integer multiple of the number of processors. This can place an unacceptably low limit on the maximum number of processors, so NLOM allows the CMT rectangle, IH by JHS, to be larger than the standard model rectangle, IH by JH. Using JHS larger than JH makes the transpose operations more expensive, but is usually faster overall than the alternative of increasing JH for the entire model. The transpose-based CMT solver appears to be scalable to at least 2048 nodes for a large enough region size. Scalability has actually been demonstrated on up to 128 T3D nodes, and up to 1024 CM5 vector units (using distributed FFTs). A semi-implicit time step, using CMT and red-black SOR to solve the Helmholtz's equations, is still the fastest available time stepping scheme for scalable ocean models.

4. NLOM PERFORMANCE

Our canonical test case is a six layer hydrodynamic finite depth 1/2 degree global model run for one year with the typical amount of I/O and data sampling. This is a relatively small problem, 512 by 288 by 6, as it has to be to allow multiple runs measuring scalability. The following table shows times for this case on several computer systems.

Table 1 – Performance of NLOM on existing HPC platforms

MACHINE	PARALLEL METHOD	NUM. NODES	TIME	MFLOP/S	SPEEDUP
Cray C90	NONE	1	1.36 hrs	355	(REAL*8)
SGI R8000	NONE	1	9.63 hrs	50	(90 MHz)
SGI R8000	AUTOTASK	2	4.91 hrs	98	1.96x 1 node
SGI R8000	AUTOTASK	4	2.47 hrs	195	1.99x 2 nodes
SGI R8000	AUTOTASK	8	1.46 hrs	331	1.69x 4 nodes
SGI R8000	AUTOTASK	16	1.08 hrs	447	1.35x 8 nodes
Cray T3D	SHMEM LIBR.	4	5.15 hrs	94	
Cray T3D	SHMEM LIBR.	8	2.70 hrs	179	1.91x 4 nodes
Cray T3D	SHMEM LIBR.	16	1.41 hrs	342	1.91x 8 nodes
Cray T3D	SHMEM LIBR.	32	0.81 hrs	596	1.74x 16 nodes
Cray T3D	SHMEM LIBR.	64	0.49 hrs	985	1.65x 32 nodes
TMC CM5E	DATA PARAL.	32	1.56 hrs	309	
TMC CM5E	DATA PARAL.	256	0.65 hrs	743	2.40x 32 nodes

On the Cray C90, Mflop/s are measured using the hardware performance monitor. Mflop/s on all other machines are based on the Cray figure, and therefore ignore any additional work performed to improve scalability.

The 1/2 degree model is relatively small. We have demonstated scalability on up to 256 nodes on a CM5E for larger problems. For example, a 32-node CM5E is 0.87 times as fast as a C90 processor on the above 1/2 degree model but at 1/8th degree a 256-node CM5E is 9.3 times faster than a C90 processor. In this case eight times as many nodes are 10.7 times faster, when the problem size is increased by a factor of sixteen.

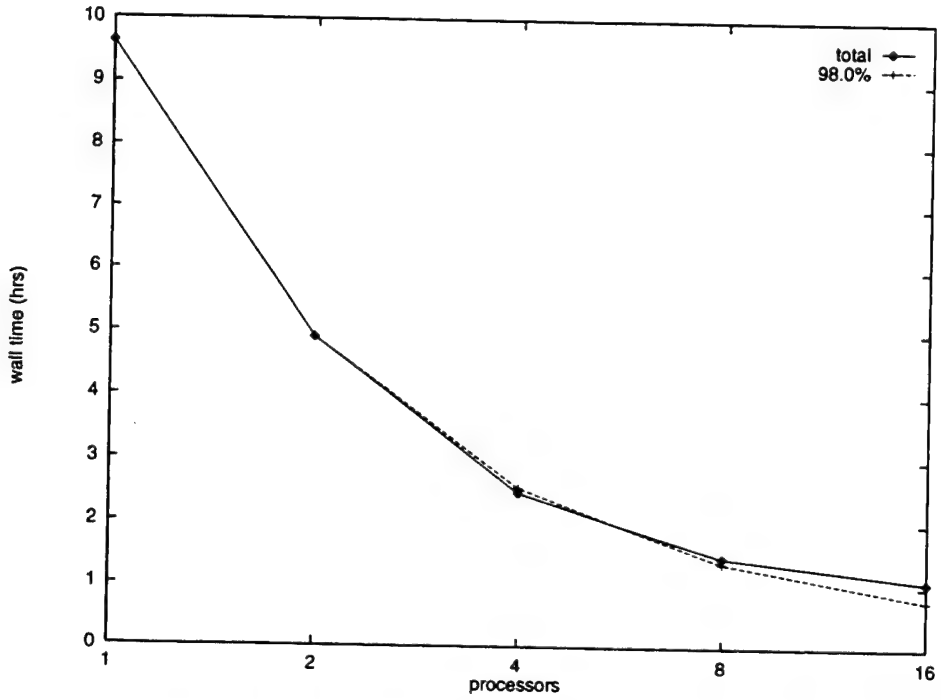

Fig. 1 – Wall time for the 1/2 degree test case on a SGI Power Challenge, compared to 98% parallelization curve

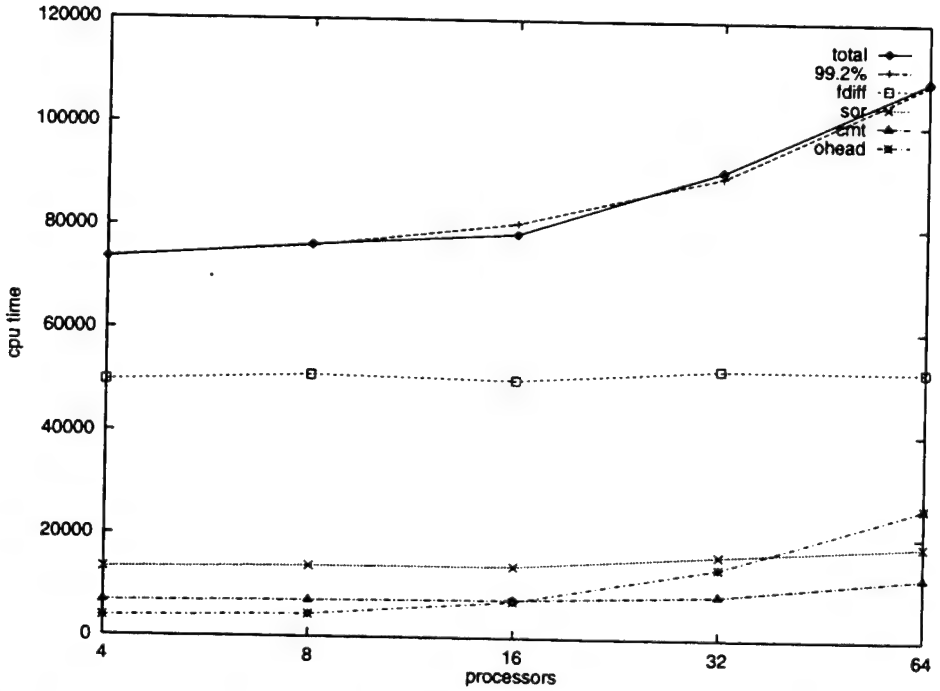

Fig. 2 – Total CPU time for the 1/2 degree test case on a Cray T3D: (a) entire model, (b) theoretical 99.2% parallel curve, (c) explicit finite difference code, (d) SOR solver, (e) CMT solver, (f) overhead.

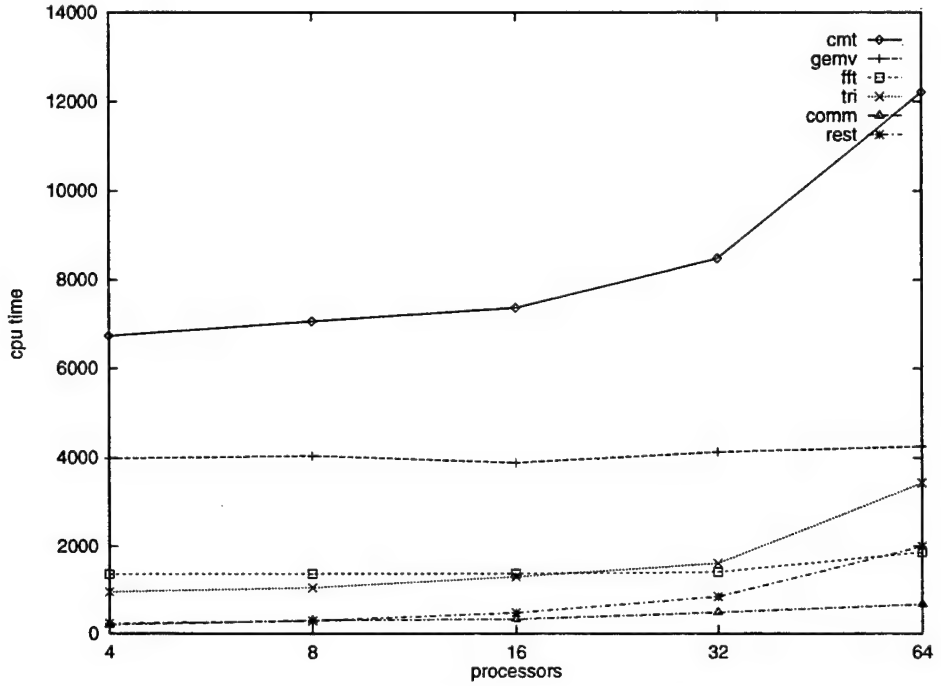

Fig. 3 – Total CPU time spent in CMT solver on a Cray T3D: (a) entire solver, (b) matrix-vector multiply, (c) FFT, (d) tridiagonal solver, (e) 2-D/1-D transposes, (f) other.

Figure 1 plots wall time against number of processors for the test case on a SGI Power Challenge. This is a close fit to Amdahl's Law for 98% parallelization. The measured performance drops below the theoretical curve for 16 processors. This could either be due to load imbalance or to a lack of memory bandwidth. Figure 2 plots total time against number of processors for the test case on a Cray T3D. Divide total time by the number of processors to get wall time. Perfect scalability would produce a horizontal curve. The model is 99.2% parallel. The explicit finite difference code is essentially 100% parallel, as expected. The SOR and CMT solvers take about 18% and 9%, respectively, of the total time and show good scalability. All the rest of the model, the overhead, represents 5% of the total on 4 nodes but 24% on 64 nodes.

Figure 3 plots total time against number of processors for the test case CMT solver on a Cray T3D. The large jump in total time on 64 nodes is primarily due to the fact that this solver is based on a 512 by 384 element array, rather than the model's 512 by 288 array. The original array can't be used on 64 nodes because 288 is not a multiple of 64. As expected, the matrix-vector multiply and FFT phases are 100% parallel. The transposes between 1D and 2D are a relatively small percentage of the total solver time, and show good scalability. The pipelined tridiagional solver has some load inbalance (from the pipelining) and many short off-chip transfers. Even so, scalability is quite good. The rest of the CMT solver has a very small operation count, but requires communication and is the factor that limits the overall scalability of the solver as a whole. Figure 4 plots total time against number of processors for the test case SOR solver on a Cray T3D. As usual, the finite difference code is 100% parallel. The halo updates are not perfectly scalable, but are a small fraction of the total even on 64 nodes. The calculation of the L2 norm of the residual requires global communication, and is the factor that limits the overall scalability of the solver as a whole. The requirement that the model be bit-for-bit reproducable on an arbitrary number of processors prevents us from using the fastest global sum algorithms.

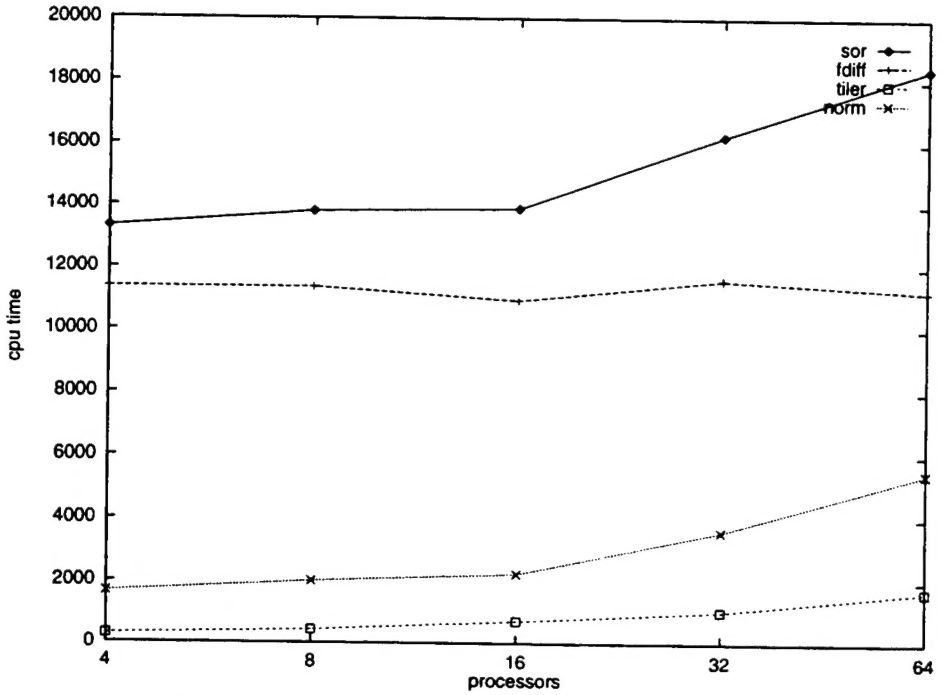

Fig. 4 – Total CPU time spent in SOR solver on a Cray T3D: (a) entire solver, (b) finite difference code, (c) halo updates, (d) l2 norm.

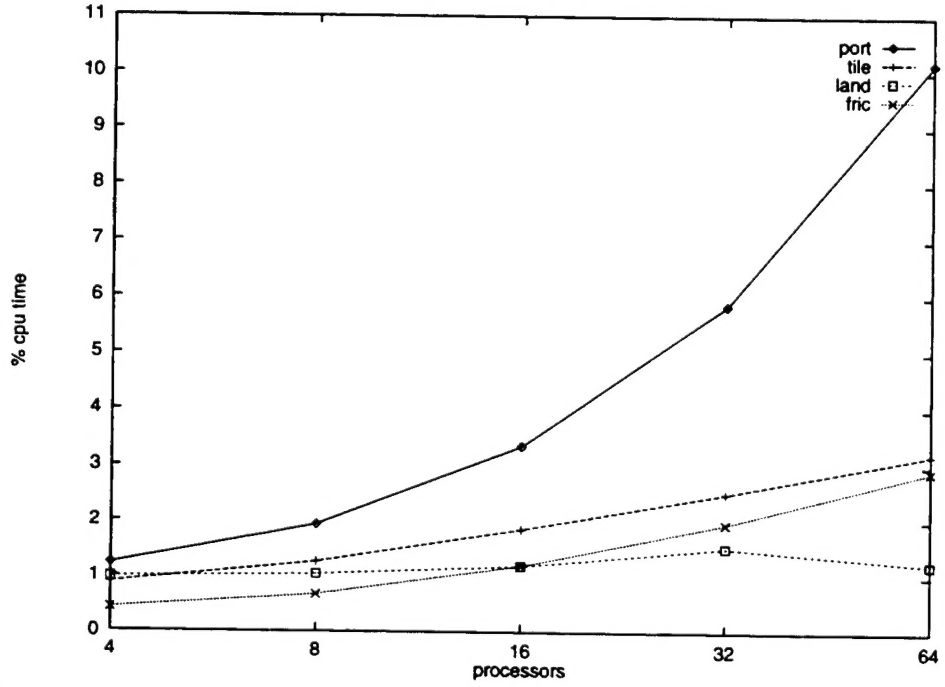

Fig. 5 – Percentage of total CPU time spent in major overhead tasks on a Cray T3D: (a) ports, (b) halo updates, (c) land boundary conditions, (d) friction patches.

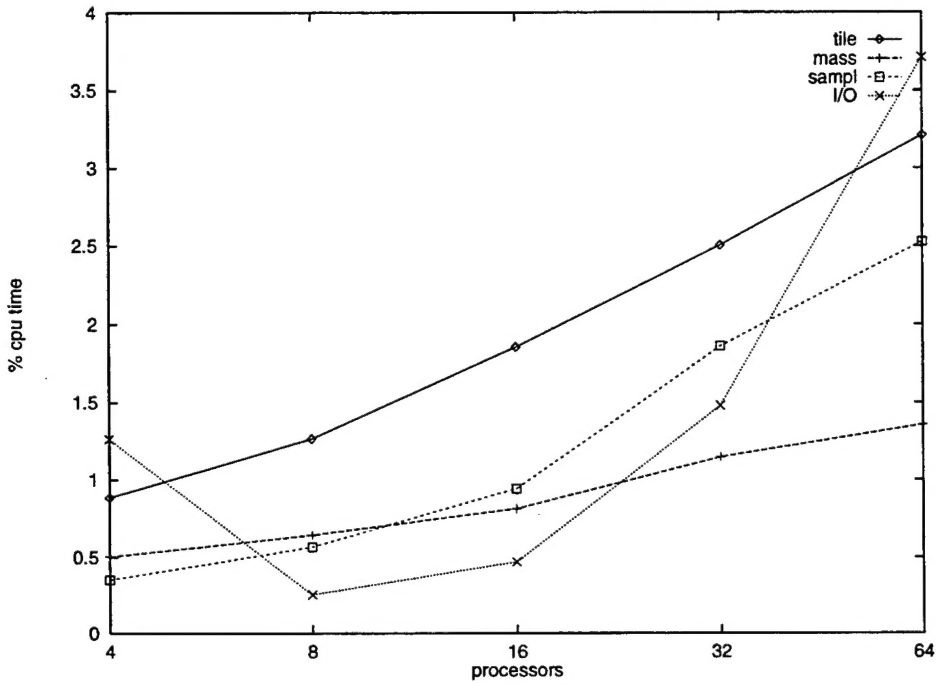

Fig. 6 – Percentage of total CPU time spent in minor overhead tasks on a Cray T3D: (a) halo updates, (b) mass calculations, (c) data sampling, (d) I/O.

Figure 5 plots the percentage of total time against number of processors for the major overhead tasks on a Cray T3D. These are all involved in boundary condition processing. Conventional boundary condition processing, for no-slip coastlines and zeroing out land areas, scales well and is only about 1% of the total time. Halo updates scale less well, but are only 3% of the total time on 64 nodes. The global region has many friction patches to control flow through straits that are not well resolved on the model grid. These are less scalable than halo updates, but also represent only 3% of total time on 64 nodes. By far the largest factor in overall scalability are the port boundary conditions. These are applied at 65N in the Atlantic to simulate the effects of the GINSea, which is not included in the model, and in particular to drive the global thermohaline circulation. The operation count for the ports is very small, but it is all "serial" code and therefore 0% scalable. Additional optimization of port routines is planned for the next release of NLOM. Figure 6 plots the percentage of total time against number of processors for the overhead tasks on a Cray T3D that are not on figure 5. The halo update curve is on both figures as a point of reference. The model's formulation of mixing requires the calculation of the region-wide mass for each layer every time step. This is similar to the L2 norm calculation in the SOR solver, with similar scalability properties, but is only 1.4% of the total time on 64 nodes. Data sampling is performed every few days throughout the simulation, to produce statistics on transport across sections, mixing in sub-regions, etcetera. The operations are not very scalable, but are relatively infrequent and represent only 2.5% of the total time on 64 nodes. The model writes out a restart record containing all prognostic variables every month. In principle, this I/O is performed in parallel on the T3D but scalability is poor. The decrease in time from 4 to 8 processors is due to the fact that smaller buffers had to be used on four processors because of memory constaints.

5. SUMMARY

The scalable NLOM implementation has achieved its design goals of running exactly the same model source code and model data files on a wide range of computer systems from many vendors. Scalability is based primarily on the tiled data parallel parallel programming paradigm. This is sufficiently general that the actual technique used on a given machine to obtain scalability can be selected at compile time from: (i) data parallel, (ii) SPMD message passing, (iii) autotasking, or (iv) SPMD message passing between multi-processor autotasked systems. Times from actual practical model runs presented in this report demonstrate good scalability across the range of processor complexes available within DoD today. As larger machines become available, problem sizes (i.e. model resolution) will increase and scalability should be possible up to at least 2048 processors.

6. ACKNOWLEDGEMENTS

This is a contribution to the 6.2 Global Ocean Prediction System Modeling Task. Sponsored by the Office of Naval Research under Program Element 62435N. Also to the Common HPC Software Support Initiative project Scalable Ocean Models with Domain Decomposition and Parallel Model Components. Sponsored by the DoD High Performance Computing Modernization Office. The benchmark simulations were performed under the Department of Defense High Performance Computing initiative, on (i) two Thinking Machines CM5Es at the Naval Research Laboratory, Washington D.C., (ii) a Cray C90 at the Naval Oceanographic Office, Stennis Space Center, Mississippi, (iii) a SGI Power Challenge at Waterways Experiment Station, Mississippi, and (iv) a Cray T3D at the Arctic Research Supercomputer Center, Alaska.

7. REFERENCES

1. Amdahl G.M. (1967), Validity of the single processor approach to achieve large scale computing capabilities. *Proc. AFIPS Spring Joint Comput. Conf.*, **30**, pp 483-485.
2. Bleck R., S. Dean, M. O'Keefe, A. Sawdey (1996). A comparison of data-parallel and message passing versions of the Miami Isopycnic Coordinate Ocean Model (MICOM). *Parallel Computing*, to appear.
3. Buzbee, B.L. F.W. Dorr, L.A. George, G.H. Golub (1971). The direct solution of the discrete Poisson equation on irregular regions. *SIAM J. Num. Anal.*, **8**, pp 722-736.
4. Dongarra J.J., J.J. Du Croz, S.J. Hammarling, R.J. Hanson (1988). An extended set of Fortran basic linear algebra subprograms. *ACM Trans. Math. Software*, **14**, pp 1-17.
5. Dongarra J.J., J.J. Du Croz, I.S. Duff, S.J. Hammarling (1990). A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software*, **14**, pp 1-17.
6. Hockney R.W. (1965). A fast direct solution of Poisson's equation using Fourier Analysis. *J.Assoc. Comput. Mach.*, **12**, pp 95-113.
7. Hockney R.W. (1996). *The science of computer benchmarking.* SIAM Press.
8. Hurlburt, H.E. & J.D. Thompson (1980). A numerical study of Loop Current intrusions and eddy shedding. *J.Phys. Oceanogr.*, **10**, pp 1611-1651.
9. Koelbel C.H., D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., M.E. Zosel (1994). *The High Performance Fortran handbook.* MIT Press.

10. Lawson, C.L., R.J. Hanson, D.R. Kincaid, F.T. Krogh (1979). Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, **5**, pp 308-323.

11. Lewis B. & D. J. Berg (1996). *Threads primer. A guide to multithreaded programming.* SunSoft Press. Prentice Hall.

12. Moore, D.R. & A.J. Wallcraft (1996). Formulation of the NRL Layered Ocean Model in Spherical Coordinates. *NRL Contractor Report* **7323–96-0005** (submitted).

13. Snir M., S.W. Otto, S. Huss-Lederman, D.W. Walker, J. Dongarra (1996). *MPI: the complete reference.* MIT Press.

14. Piacsek, S.A., A.J. Wallcraft (1993). Initial experiences with the Connection Machine at NRL. *NRL Technical Note* **73-5089-03**.

15. Wallcraft, A.J (1991). The Navy Layered Ocean Model users guide. *NOARL Report* **35**. Naval Research Laboratory, Stennis Space Center, MS, 21 pp.